系统调用的添加与进程调度算法的修改

# 实验目的

## 建立对系统调用接口的深入认识，掌握系统调用的基本过程，能完成系统调用的全面控制

## 掌握进程调度的过程，了解进程调度算法并且能够对其修改

# 实验内容

此次试验的基本内容是：

1. 在Linux 0.11上添加第二种进程调度算法
2. 添加一个系统调用实现在两种调度算法之间进行切换
3. 编写适当的应用来验证算法的正确性，重点是体现出两种调度算法的区别

# 实验报告

在实验报告中回答如下问题：

1. 用文字简要描述向Linux 0.11添加一个系统调用call()的步骤
2. 用文字简要描述你所实现的系统调用的算法，并列出实现算法的关键代码

# 评分标准

1. 成功添加系统调用占40%
2. 成功实现调度算法占40%
3. 成功用应用验证占10%
4. 实验报告占10%

# 实验提示

操作系统实现系统调用的基本过程是：

1. 应用程序调用库函数；
2. 库函数将系统调用号存入EAX，然后通过中断使系统进入内核态；
3. 内核中的中断处理函数根据系统调用号，调用对应的内核函数；
4. 系统调用完成相应功能，将返回值存入EAX，返回到中断处理函数；
5. 中断处理函数返回到库函数中；
6. 库函数将EAX返回给应用程序。

## 应用程序调用系统调用

一般来说，系统调用的库函数和普通的自定义函数在使用和定义上并没有什么区别，不同之处体现在调用之后，一般的自定义函数被调用后通过call指令调到该函数的地址继续运行。而系统调用函数会通过中断的方式进入系统内核调用相应的内核函数执行。进入内核的过程是：

1. 把系统调用的编号存入EAX
2. 把函数参数存入其他通用寄存器
3. 触发0x80号中断（int 0x80）

0.11的lib目录下有一些已经实现的系统调用，实现它们的原因是内核启动后会做些初始化工作，进入用户态后的初始化工作要用到很多系统调用，这些系统调用就是为内核的初始化服务。首先研究一下close()这个系统调用：

#define \_\_LIBRARY\_\_

#include <unistd.h>

\_syscall1**(**int**,**close**,**int**,**fd**)**

其中\_syscall1是一个宏，在include/unistd.h中定义。将\_syscall1(int,close,int,fd)进行宏展开，可以得到：

int close**(**int fd**)** \

**{** \

long \_\_res**;** \

\_\_asm\_\_ volatile **(**"int $0x80" \

**:** "=a" **(**\_\_res**)** \

**:** "0" **(**\_\_NR\_close**),**"b" **((**long**)(**a**)));** \

**if** **(**\_\_res **>=** 0**)** \

**return** **(**type**)** \_\_res**;** \

errno **=** **-**\_\_res**;** \

**return** **-**1**;** \

**}**

这就是系统调用的定义。它首先将宏\_\_NR\_close存入EAX，将参数fd存入EBX，然后进行0x80中断调用。调用返回后，从EAX取出返回值，存入\_\_res，在通过对\_\_res的判断决定传给调用者什么样的返回值。其中\_\_NR\_close就是系统调用的编号，在include/unistd.h中定义：

#define \_\_NR\_close 6

所以添加系统调用时需要修改include/unistd文件，使其包含\_\_NR\_xxx，其中xxx是你要添加的系统调用的名字。而在应用程序中要有：

#define \_\_LIBRARY\_\_

#include <unistd.h>

\_syscall1**(**int**,**close**,**int**,**fd**)**

其中第三行\_syscall后面的数字取决于你传递几个参数，具体参考unistd.h中的\_syscall0、\_syscall1、\_syscall2、\_syscall3。宏的参数也需要自己决定。

## 从“int 0x80”进入内核函数

Int 0x80触发后，接下来就是内核的中断处理了。首先了解0.11处理0x80号中断的过程。

在内核初始化时，主函数（在init/main.c中）调用了sched\_init()初始化函数：

time\_init**();**

sched\_init**();**

buffer\_init**(**buffer\_memory\_end**);**

hd\_init**();**

floppy\_init**();**

sched\_init()在kernel/sched.c中定义为：

void sched\_init**(**void**)**

**{**

...

set\_system\_gate**(**0x80**,&**system\_call**);**

**}**

set\_system\_gate是个宏，在include/asm/system.h里被定义为：

#define set\_system\_gate(n,addr) \

\_set\_gate(&idt[n],15,3,addr)

\_set\_gate的定义是：

#define \_set\_gate(gate\_addr,type,dpl,addr) \

\_\_asm\_\_ ("movw %%dx,%%ax\n\t" \

"movw %0,%%dx\n\t" \

"movl %%eax,%1\n\t" \

"movl %%edx,%2" \

: \

: "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \

"o" (\*((char \*) (gate\_addr))), \

"o" (\*(4+(char \*) (gate\_addr))), \

"d" ((char \*) (addr)),"a" (0x00080000))

\_set\_gate的作用就是填写IDT表的中断描述符，将system\_call函数的地址填写到第0x80号中断描述符的地方，之后一执行int 0x80指令，就会跳到system\_call的位置执行。

接下来看一下system\_call，该函数是纯汇编函数，在kernel/system\_call.s里面定义如下：

\_system\_call:

cmpl $nr\_system\_calls-1,%eax

ja bad\_sys\_call

push %ds

push %es

push %fs

pushl %edx

pushl %ecx # push %ebx,%ecx,%edx as parameters

pushl %ebx # to the system call

movl $0x10,%edx # set up ds,es to kernel space

mov %dx,%ds

mov %dx,%es

movl $0x17,%edx # fs points to local data space

mov %dx,%fs

call \_sys\_call\_table(,%eax,4)

pushl %eax

movl \_current,%eax

cmpl $0,state(%eax) # state

jne reschedule

cmpl $0,counter(%eax) # counter

je reschedule

...

首先我们需要关心的是cmpl $nr\_system\_calls-1,%eax，这句是将eax里面的系统调用号和常数nr\_system\_calls相比较，nr\_system\_calls代表的是内核系统调用的个数，如果eax比nr\_system\_calls大，则不往下执行，立即返回用户态，所以添加了新的系统调用就要修改nr\_system\_calls的值，这个值的定义就在system\_call.s中。

call \_sys\_call\_table(,%eax,4)之前是一些压栈保护，修改段选择符为内核段，之后判断是否需要重新调度，这些都跟实验没有直接关系，此处只关心call \_sys\_call\_table(,%eax,4)这一句，这一句的寻址方式是：

call \_sys\_call\_table+4\*%eax

\_sys\_call\_table是个数组的地址，它在include/linux/sys.h中被声明：

fn\_ptr sys\_call\_table**[]** **=** **{** sys\_setup**,** sys\_exit**,...}**

这些函数与用户态的系统调用是相互对应的，比如用户态的setup()函数对应着函数表里面的sys\_setup()函数，同时函数表里面的函数真正实现了系统调用的功能，增加系统调用，必须在这个函数表中添加对应的函数引用，当然该函数在此函数表中的位置必须与\_\_NR\_xxx的值相对应。同时还要在sys.h中添加函数声明extern int sys\_xxx()，不然会出错。

## 系统调用的实现以及调度算法的选择

系统调用的真正功能的实现是在名为sys\_xxx()的内核函数里面，这里我们需要完成的功能就是实现对两种调度算法的选择，这一点不复杂，方法也很多，不再赘述。

非常重要的一部分就是调度算法的实现，首先是调度算法的选择，调度算法有很多，这里提供一种选择FIFO，使越早被替换下来的进程越早的再次被调度到。这种方式的实现方法可以用队列来实现，定义队列中的结构体，结构体中包含2个元素，类型是int和指向结构体的指针next，next是为了实现队列，int代表此实体代表的task\_struct在整个task\_struct数组中的下标，在schedule()函数中的调度算法的核心是：

**if** **((\***p**)->**state **==** TASK\_RUNNING **&&** **(\***p**)->**counter **>** c**)**

c **=** **(\***p**)->**counter**,** next **=** i**;**

这句的意思是在所有的状态是TASK\_RUNNING的进程中，选择一个counter值最大的进程切换，而现在我们所做的工作就是在队列中查找找到第一个状态为TASK\_RUNNING的进程切换并把当前进程的实体插入队列尾部。

调度算法可以用一个新的函数实现，也可以在原schedule()函数中添加，以我的经验，用新的函数实现会发生编译错误，尽管这样实现看起来是比较规整的。

## 编写用户态程序验证

根据自己选择的调度算法设计适当的程序去验证，重点是体现出两种调度算法的区别。

这里给出一个FIFO算法的验证思路，可以编写用户态程序分别创建几个进程，让他们分别打印不同的内容，比如创建3个进程，第一个进程循环打印A，第二个进程循环打印B，第三个进程循环打印C，那么如果调度算法是FIFO算法，字母的打印顺序是不变的，原来调度算法的打印顺序则不一定。